Azzolini Riccardo 2019-09-30

Ereditarietà

1 Riferimento alla superclasse

All'interno dei metodi di una sottoclasse, è possibile riferirsi alla superclasse usando la parola chiave `super`.

Ciò è utile, ad esempio, quando si ridefinisce un metodo nella sottoclasse e, al suo interno, si vuole invocare il metodo originale della superclasse:

```
class SuperClasse {
    public void metodo() {
        // ...
    }
}

class SottoClasse extends SuperClasse {
    public void metodo() {
        // ...
        super.metodo();
        // ...
    }
}
```

Infatti, se si scrivesse semplicemente `metodo()`, questa sarebbe una chiamata ricorsiva al metodo ridefinito stesso.

In un metodo ridefinito scritto in questo modo:

- `super.metodo()` effettua le operazioni necessarie sui dati membro ereditati dalla superclasse (alcuni dei quali potrebbero essere privati, quindi tale chiamata potrebbe essere l'unico modo di accedervi);
- la parte restante del metodo effettua le operazioni specializzate per la sottoclasse.

Java non permette di riferirsi alla superclasse della superclasse: `super.super` è un errore di sintassi.

2 Costruttori e finalizzatori di una sottoclasse

Ogni oggetto di una sottoclasse è composto da:

- un oggetto della superclasse;
- gli elementi aggiunti nella sottoclasse.

Per questo, all'inizio dell'esecuzione di un costruttore della sottoclasse viene *sempre* chiamato un costruttore della superclasse.

- Si può effettuare una chiamata esplicita, con la sintassi `super(argomenti)`, scritta come prima istruzione del costruttore.
- In assenza di una chiamata esplicita, viene chiamato implicitamente il costruttore per default della superclasse.

Nel finalizzatore di una sottoclasse, è buona norma chiamare, come ultima istruzione, il finalizzatore della superclasse (`super.finalize()`), ma esso non viene invocato in automatico (cosa che invece avviene per i distruttori in C++).

3 Gerarchia di ereditarietà

L'ereditarietà è una relazione transitiva: se `A` è superclasse di `B` e `B` è superclasse di `C`,

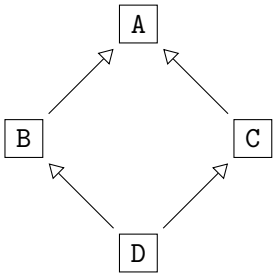
```
class A { /* ... */ }
class B extends A { /* ... */ }
class C extends B { /* ... */ }
```

allora `A` è superclasse di `C`, cioè `C` eredita (indirettamente) da `A`.

Per le classi, Java supporta solo l'*ereditarietà singola*: una classe può ereditare da una sola classe. Di conseguenza, la gerarchia di ereditarietà è un albero (n-ario, perché ogni classe può avere un numero n qualsiasi di sottoclassi), la cui radice è la classe `Object`, dalla quale ereditano (direttamente o indirettamente) tutte le altre.

I progettisti del linguaggio Java hanno scelto di non permettere l'*ereditarietà multipla* per le classi perché essa presenta alcuni problemi:

- In una gerarchia come



i dati e i metodi definiti nella classe `A` sono ereditati sia da `B` che da `C`, quindi `D` ne eredita due copie. Così, oltre allo spreco di memoria, si ha un'ambiguità quando si accede a tali dati/metodi su un'istanza di `D`: l'accesso si potrebbe riferire alla versione di `B`, oppure a quella di `C` (o anche a quella di `A`, se è diversa dalle altre due perché il dato/metodo è stato ridefinito in `B` e/o in `C`). Ci sono dei modi per disambiguare, ma essi introducono un'ulteriore complessità nel linguaggio.

- La ricerca dei metodi da eseguire è più complessa, dato che possono esserci più percorsi lungo cui risalire la gerarchia (perché essa non è necessariamente un albero).

4 Metodi e classi `final`

La parola chiave `final` permette, in generale, di fissare il significato di un'entità di programma, impedendo che esso venga ridefinito:

- i metodi `final` non possono essere ridefiniti nelle sottoclassi;
- le classi `final` non possono essere specializzate, cioè non è possibile creare sottoclassi che ereditano da tale classe.

I metodi `static` sono anche `final`, quindi non possono solitamente essere ridefiniti.

I metodi `private`, invece, possono sempre essere ridefiniti, anche se sono `final` (o `static`), perché non sono visibili nelle sottoclassi.

5 Metodi e classi `abstract`

- I metodi `abstract` non hanno un corpo, quindi devono per forza essere ridefiniti nelle sottoclassi.
- Le classi `abstract` non possono essere istanziate. Per utilizzarle, bisogna creare delle sottoclassi concrete (non `abstract`) e istanziare quelle.

Una classe *deve* essere dichiarata `abstract` se contiene almeno un metodo `abstract`, ma può essere dichiarata `abstract` anche se contiene solo metodi concreti (ciò può essere utile per realizzare una gerarchia di classi nella quale tutte le istanze devono essere delle sottoclassi: ad esempio, data una classe `Studente` con due sottoclassi, `StudenteTriennale` e `StudenteMagistrale`, non ha senso creare studenti "generici", che non sono né della triennale né della magistrale, quindi è meglio che `Studente` sia dichiarata `abstract`, anche se magari non contiene metodi astratti).

Nonostante non si possano istanziare, le classi astratte possono contenere dati membro, anche privati, quindi devono comunque avere dei costruttori, per inizializzare tali dati quando si costruiscono oggetti delle sottoclassi.

6 Interfacce

Le interfacce sono effettivamente degli "scheletri" di classi, in cui:

- tutti i metodi sono `public abstract`
- tutti i dati sono `public static final` (cioè costanti)

anche se non è scritto esplicitamente.

Tra le interfacce, è permessa l'ereditarietà multipla: un'interfaccia può ereditare da un numero qualsiasi di altre interfacce. Inoltre, una classe può implementare ("ereditare" da) più interfacce.

Poiché le interfacce non contengono metodi concreti o dati non costanti, non si hanno gli stessi problemi associati all'ereditarietà multipla tra classi.

7 Polimorfismo

Un oggetto di una sottoclasse può essere trattato come un oggetto di una qualsiasi delle sue superclassi. Viceversa, se si tratta un oggetto di una superclasse come un oggetto di una delle sue sottoclassi si possono avere errori.

Per questo, in Java, un riferimento a un oggetto di una sottoclasse può essere assegnato a un riferimento a un oggetto di una delle sue superclassi, ma non si può fare il contrario (è un errore in fase di compilazione).

Siccome il riferimento a un oggetto di una superclasse è una variabile, può puntare a oggetti diversi durante l'esecuzione di un programma. Tali oggetti possono anche essere di sottoclassi diverse, cioè possono assumere "molte forme", quindi questo meccanismo viene chiamato **polimorfismo**.

7.1 Vantaggi e svantaggi

Il polimorfismo, insieme all'ereditarietà, è un meccanismo molto potente:

- permette il riuso del software;
- permette di usare sempre il metodo o dato membro corretto per ogni oggetto, tramite il **binding dinamico** (**late binding**);
- favorisce una vera estendibilità del software;
- favorisce la modificabilità del software.

Tuttavia, l'uso del polimorfismo diminuisce:

- la leggibilità del software, perché non è possibile sapere esattamente quale metodo (di quale sottoclasse) verrà eseguito semplicemente leggendo il codice;
- la verificabilità del software, perché la stessa istruzione potrebbe dare errori solo in alcuni casi, quindi si ha una perdita di *consistenza* (la proprietà di un programma di funzionare sempre correttamente o dare sempre errore).

7.2 Binding dinamico

Quando si chiama un metodo su un oggetto, non è detto che il metodo effettivamente eseguito sia stato dichiarato nella classe a cui appartiene tale oggetto: in caso contrario, viene eseguito il metodo con la stessa segnatura[1] dichiarato nella superclasse più vicina.

In generale, non è noto al momento della compilazione quale metodo verrà eseguito: il collegamento (binding) tra il nome del metodo e il metodo effettivamente eseguito è "tardivo" (late), in quanto viene determinato solo durante l'esecuzione, in base al tipo di oggetto puntato effettivamente dal riferimento al momento della chiamata.

8 Conversioni di tipo dei riferimenti

8.1 Da sottoclasse a superclasse

Un riferimento a un oggetto di una sottoclasse può essere convertito *implicitamente* a un riferimento a un oggetto di una delle sue superclassi. In tal caso, i metodi e dati che si possono usare sono quelli

- originari della superclasse, non ridefiniti nella sottoclasse
- ridefiniti nella sottoclasse

[1] *Osservazione*: La segnatura non comprende il tipo restituito, quindi esso non viene considerato per la scelta del metodo da eseguire.

ma *non* quelli aggiunti nella sottoclasse, non esistenti nella superclasse. Infatti, il compilatore non può garantire che un riferimento a un oggetto della superclasse punti concretamente a un oggetto della sottoclasse, quindi i dati/metodi aggiunti in quest'ultima potrebbero non essere disponibili.

```
SottoClasse rifSotto = new SottoClasse();
SuperClasse rifSopra = rifSotto; // Conversione implicita

rifSopra.metodoNonRidefinito();
// OK: eseguito il metodo originale di SuperClasse

rifSopra.metodoRidefinito();
// OK: eseguito il metodo ridefinito in SottoClasse

rifSopra.metodoAggiunto();
// Errore in compilazione
```

Con una conversione di tipo esplicita (cast), si ottengono gli stessi risultati.

```
SottoClasse rifSotto = new SottoClasse();

((SuperClasse)rifSotto).metodoNonRidefinito();
// OK: eseguito il metodo originale di SuperClasse

((SuperClasse)rifSotto).metodoRidefinito();
// OK: eseguito il metodo ridefinito in SottoClasse

((SuperClasse)rifSotto).metodoAggiunto();
// Errore in compilazione
```

8.2 Da superclasse a sottoclasse

Un riferimento a un oggetto di una superclasse può essere convertito *solo esplicitamente* (cast) a un riferimento a un oggetto di una delle sue sottoclassi.

```
SuperClasse rifSopra = new SottoClasse();
SottoClasse rifSotto;

rifSotto = rifSopra;
// Errore in compilazione

rifSotto = (SottoClasse)rifSopra;
// OK
```

```
((SottoClasse)rifSopra).metodoNonRidefinito();
// OK: eseguito il metodo originale di SuperClasse

((SottoClasse)rifSopra).metodoRidefinito();
// OK: eseguito il metodo ridefinito in SottoClasse

((SottoClasse)rifSopra).metodoAggiunto();
// OK: eseguito il metodo aggiunto in SottoClasse
```

Se l'oggetto a cui punta il riferimento non è effettivamente un'istanza di tale sottoclasse, viene sollevata un'*eccezione in fase di esecuzione*.

```
SuperClasse rifSopra = new SuperClasse();
SottoClasse rifSotto;

rifSotto = (SottoClasse)rifSopra;
// Eccezione in esecuzione

((SottoClasse)rifSopra).metodoNonRidefinito();
// Eccezione in esecuzione

((SottoClasse)rifSopra).metodoRidefinito();
// Eccezione in esecuzione

((SottoClasse)rifSopra).metodoAggiunto();
// Eccezione in esecuzione
```

Prima di effettuare una conversione di questo tipo, bisogna quindi essere sicuri che l'oggetto sia effettivamente un'istanza della sottoclasse. A tale scopo, si può utilizzare per esempio l'operatore `instanceof`:

```
SuperClasse rifSopra;
// ...
if (rifSopra instanceof SottoClasse) {
    SottoClasse rifSotto = (SottoClasse)rifSopra;
    ((SottoClasse)rifSopra).metodoNonRidefinito();
    ((SottoClasse)rifSopra).metodoRidefinito();
    ((SottoClasse)rifSopra).metodoAggiunto();
}
```